Programación avanzada en Python

Lección 2: Controladores de flujo y colecciones de datos

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

Controladores de flujo y colecciones de datos

PRESENTACIÓN Y OBJETIVOS

En este segundo capítulo vamos a ver los principales controladores de flujo que podemos utilizar en Python, además veremos algunas de sus diferencias más importantes. También veremos como trabajar con tuplas, conjuntos y diccionarios que son colecciones de datos muy útiles en el desarrollo de programas en Python. Por último, veremos como introducir y mostrar datos por pantalla.

Objetivos

- En esta lección aprenderás a:
- Controladores de flujo.
- Como trabajar con tuplas, conjuntos y diccionarios
- Introduccir y mostrar datos de nuestros programas

1. CONTROLADORES DE FLUJO

Los controladores de flujo determinan el flujo de ejecución de un programa basándose en algunas condiciones. Las condiciones deciden qué línea de código se ejecuta o no. Por lo tanto, todos los controladores de flujo son tomadores de decisiones. Ayudan a ejecutar algunas acciones particulares de acuerdo con algunos criterios.

1.1 Sentencia IF

La sentencia If es el controlador de flujo más común y puede comprobar la condición con la palabra clave 'if'

Sintaxis para un if simple:

if <condición>:

    Sentencia 1

Comprueba la condición y si es verdadera, se ejecutará la sentencia 1.

El diagrama de flujo de la sentencia if es

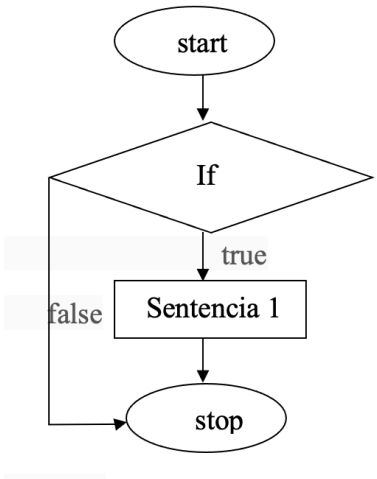

Figura 1.1: Diagrama de flujo if

Veamos un ejemplo:

```
number = 5
if number < 10 :
  print("number is less than 10")
```

```
number is less than 10
```

Los dos puntos después de la condición "if" son obligatorios y la declaración debe tener una sangría adecuada.

1.1.2 if-else

Cuando la condición es falsa, podemos utilizar la parte else si hay una sentencia que ejecutar.

Sintaxis:

If condición:

    sentencia 1

else:

    sentencia 2

Si la condición es falsa, se ejecutará la sentencia 2.

Es necesario poner dos puntos después de la condición else y la sentencia debe tener la sangría adecuada.

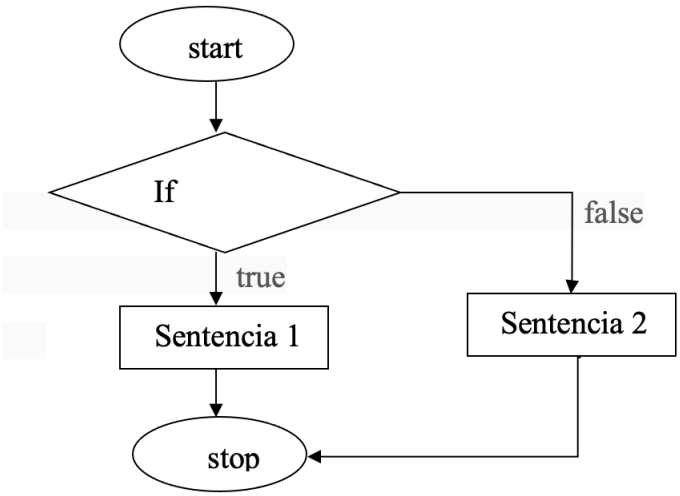

Figura 1.2: Diagrama de flujo if-else

Veamos un ejemplo:

```
number = 12
if number < 10 :
  print("number is less than 10")
else:
  print("number is greater than 10")
```

```
number is greater than 10
```

1.1.3 If – elif – else

Si hay varias condiciones que comprobar, se puede utilizar la sentencia if- elif- else. 'elif' es una palabra clave utilizada en lugar de else if, que de nuevo comprueba una condición en el caso de que sea falsa la sentencia if.

Si todas las condiciones fallan, la parte else se ejecutará o imprimirá algo no válido.

Sintaxis:

condición if:

    declaración 1

condición elif:

    declaración 2

elif .....

    .....

else:

    imprimir condición no válida

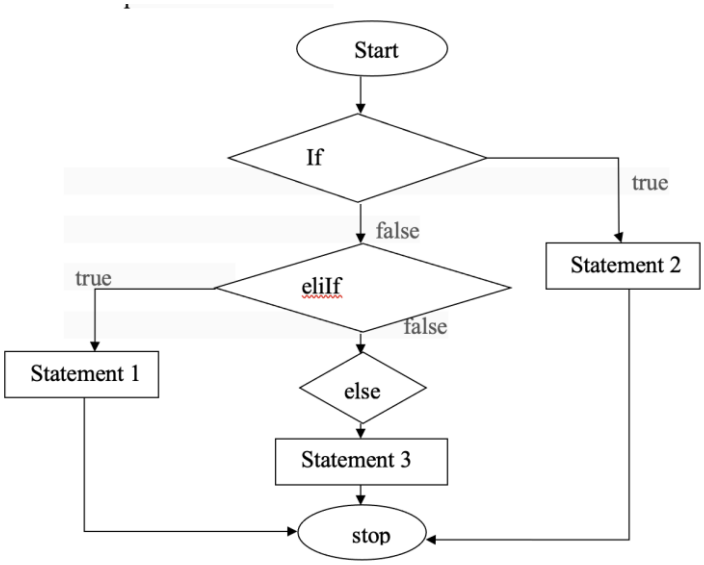

Figura 1.3: Diagrama de flujo if-elif-else

Ejemplo:

```
number = 12
if number < 10 :
  print("number is less than 10")
elif number > 15:
  print("number is greater than 15")
else:
  print("number is in between 10 and 15")
```

```
number is in between 10 and 15
```

En este programa se comprueba la condición de la parte if y si es falsa, entonces se comprueba la condición en la parte elif la cual, también es falsa. Así que se ejecuta la parte else e imprime el mensaje.

1.2 Sentencia While

El controlador **'if'**, ejecuta una condición o un conjunto de condiciones únicamente una vez. La sentencia **while** puede comprobar varias veces una condición particular para un conjunto de valores.

Es un bucle que comprueba la condición continuamente para un conjunto de valores, por lo que necesita algunos parámetros obligatorios:

- **Condición**- comprueba una condición para todo el conjunto de valores. Si la condición es falsa, entonces saldrá del bucle. Si la condición es verdadera, entonces el bucle continuará hasta que sea falsa
- **Variable contador** - que decide cuántas veces iterará el bucle y debe ser incrementada después de cada iteración, de lo contrario el bucle se vuelve infinito, incluso si la condición es falsa o no.
- **Break** - incluso si la condición es verdadera, a veces el bucle debe de terminar después de realizar algunas acciones. La sentencia break puede ser usada para eso y saldrá del bucle cuando la dicha sentencia se ejecute.

Sintaxis:

La condición se da con la palabra clave while y antes de eso se inicializa la variable contador. Dentro del bucle se escriben las sentencias a ejecutar y se incrementa la variable contador para continuar el bucle hasta que la condición se vuelva falsa.

```
Variable_contador = 0
while condition:
    statement1
    ……….....
    Variable_contador += 1
```

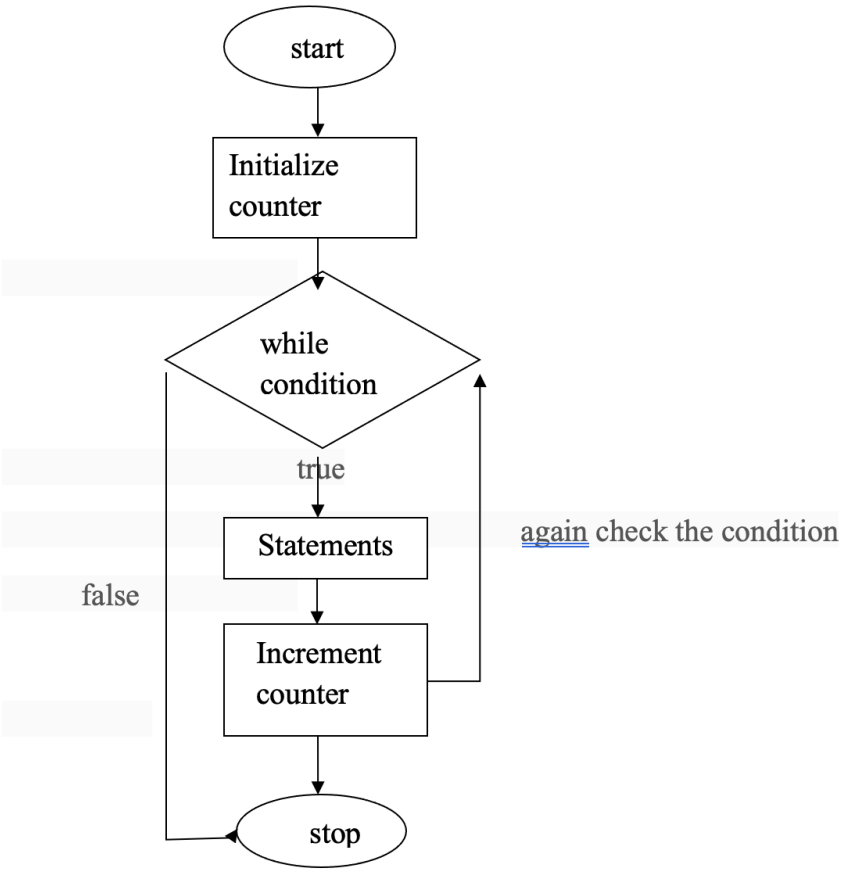

Figura 1.4: Diagrama de flujo while

Veamos un ejemplo:

```
num = 1
while num <= 5:
  print(num)
  num += 1
```

```
1
2
3
4
5
```

En el anterior ejemplo, *num* es la variable contador y el bucle se ejecuta hasta que *num* se convierte en 6.

Comprobemos qué ocurre si el contador no se incrementa:

```
num = 1
while num <= 5:
  print(num)
```

```
1
1
1
1
1
1
1
```

En este ejemplo, el bucle se vuelve infinito y nunca termina, la condición es siempre verdadera para 1. Esta es la importancia de incrementar la variable contador o poner la sentencia break.

```
num = 1
while num <= 5:
  print(num)
  if num == 3:
    break
  num += 1
```

```
1
2
3
```

Aquí se imprime sólo hasta el 3 porque la sentencia break pone una condición concreta y se saltan el resto de condiciones.

1.3 Sentencia For

Los bucles while y for tienen el mismo propósito, se utilizan para ejecutar un conjunto de sentencias dada una condición. La mayor diferencia es que hay un incremento automático del bucle dentro del bucle for en lugar de la variable contador que debe ser incrementada manualmente en el bucle while.

Para incrementar automáticamente la variable del contador, el bucle for toma la ayuda de una función incorporada llamada range (). Esta función devuelve una secuencia de números que se especifican en el parámetro. La función puede tener hasta 3 parámetros.

Sintaxis:

range(n) - devuelve un conjunto de números de 0 a n

range(inicio, fin) - devuelve un conjunto de números entre el inicio y el fin

range(start, end, step) - devuelve un conjunto de números entre el inicio y el final saltando con el valor de *step*.

Veamos un ejemplo de los números que devuelve range () mediante una lista:

```
list(range(8))
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
list(range(5,15))
```

```
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

```
list(range(5,100,20))
```

```
[5, 25, 45, 65, 85]
```

Sintaxis para el bucle for:

```
for variable in range(n):
    statement 1
    ……..
```

Las declaraciones en el bucle **'for'** se ejecuta n iteraciones para la 'variable'. La operación de incremento no quiere sumar manualmente. Incrementa automáticamente el valor de la variable hasta n especificado en la función de *range()*.

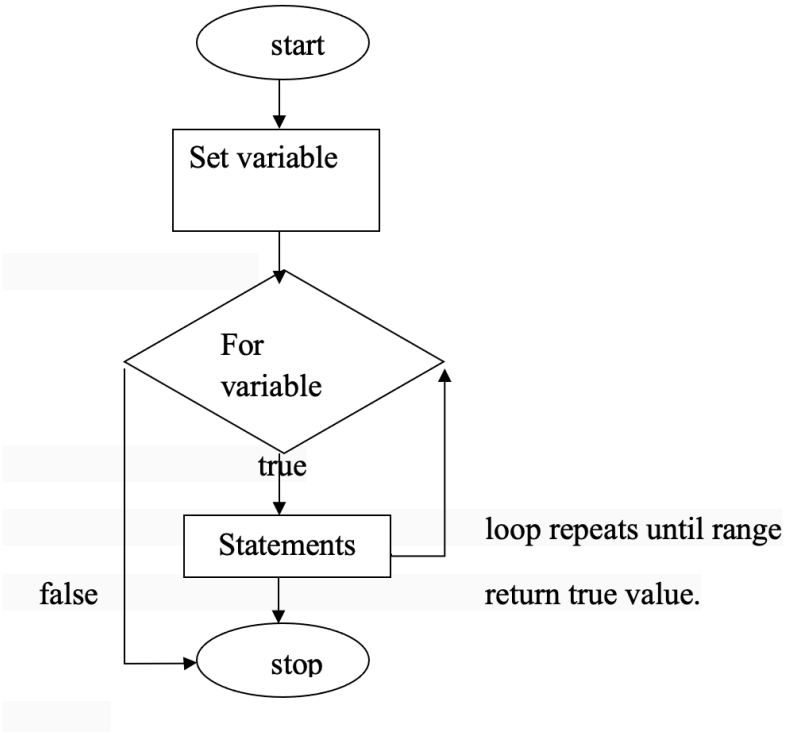

Figura 1.5: Diagrama de flujo for

```
for value in range(5) :
  print(value)
```

```
0
1
2
3
4
```

El valor se incrementa automáticamente hasta el 5 especificado en la función de range().

El conjunto de números pares puede imprimirse fácilmente utilizando el bucle for configurando la función range() como:

```
for value in range(0,10,2) :
  print(value)
```

```
0
2
4
6
8
```

También se pueden decrementar el contador:

```
for value in range(5,0,-1) :
  print(value)
```

```
5
4
3
2
1
```

1.3.1 Bucle for anidado

Un bucle for puede ser anidado dentro de otro bucle for y así sucesivamente.

Sintaxis:

```
for variable1 in range(n):
    statements
    for variable 2 in range(m):
        statements
```

Veamos un ejemplo:

```
for i in range(3):
  for j in range(2):
    print(i)
  print("\n")
```

```
0
0


1
1


2
2
```

El bucle exterior se ejecuta 3 veces y el bucle interior se ejecuta 2 veces

1.3.2 Bucles anidados complejos:

Los bucles while y for pueden anidarse en cualquier orden. El bucle exterior puede ser while/for y el interior for/while respectivamente.

Lo importante es que la indentación sea adecuada.

```
counter = 1
while counter<=3:
  for i in range(3):
    print(i+1 ,"*", counter, "=" ,(i+1)*counter)
  counter += 1
```

El bucle externo es while y el interno es for. Imprime la tabla de multiplicar hasta 3 para 1, 2 y 3:

```
1 * 1 = 1
2 * 1 = 2
3 * 1 = 3
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
```

Podemos imprimir la tabla de multiplicación completa cambiando los parámetros.

Usando un bucle for, podemos recorrer una lista:

```
shop_list = ['apple',10,'orange',20,'tomato',5,'banana',10]
```

```
for item in shop_list:
  print(item)
```

```
apple
10
orange
20
tomato
5
banana
10
```

2. TUPLAS

La tupla es una estructura de datos que recoge un conjunto de elementos de diferentes tipos de datos. Una de las principales diferencias de las tuplas con respecto a otros tipos de datos es que, una vez creadas, no se pueden modificar. Los elementos de las tuplas se definen con paréntesis.

Sintaxis:

tuple_name = (item1. Item2… item n)

Ejemplo

```
fruits =("apple", "orange", "banana")
```

Se puede acceder a los elementos por su índice a partir de cero

```
fruits[0]
```

```
'apple'
```

La tupla es inmutable. Así que las operaciones de inserción, borrado y modificación no funcionan.

Cuando se intenta modificar un elemento:

```
fruits[1] = "potato"
```

Entonces se mostrará un error como:

```
TypeError: 'tuple' object does not support item assignment
```

Al intentar añadir un nuevo elemento:

```
fruits.append("potato")
```

```
AttributeError: 'tuple' object has no attribute 'append'
```

Al intentar eliminar un elemento:

```
del fruits[0]
```

```
TypeError: 'tuple' object doesn't support item deletion
```

El concepto de indexación y slicing sólo funcionará para las tuplas.

El slicing se puede realizar como:

```
fruits[1:2]
```

```
('orange',)
```

Las tuplas pueden contener los elementos con diferentes tipos de datos:

```
fruits = ("apple", 5,"orange",10, "banana",8)
```

2.1 Conversión entre lista y tupla

Una tupla se puede convertir en lista llamando a list (). El parámetro de la función es el nombre de la tupla.

```
new_list = list(fruits)
```

```
print(new_list, type(new_list))
```

```
['apple', 'orange', 'banana'] <class 'list'>
```

El contenido de la tupla ahora se convierte en lista y se pueden hacer otras operaciones con los datos fácilmente.

También el contenido de la lista se puede convertir en tupla.

```
new_tuple = tuple(new_list)
```

```
print(new_tuple, type(new_tuple))
```

```
('apple', 'orange', 'banana') <class 'tuple'>
```

3. CONJUNTOS

Los conjuntos es otra estructura de datos que recoge los elementos dentro de corchetes

La principal característica de los conjuntos es que son desordenados y la indexación no funciona.

Sintaxis:

nombre_conjunto = {elemento 1, elemento 2... elemento n}

Ejemplo:

```
country = {"India","USA","UAE"}
```

Vamos a comprobar el tipo de dato:

```
print(type(country))
```

```
<class 'set'>
```

Si se intenta acceder a los elementos del conjunto, se muestra un error

```
country[0]
```

```
TypeError: 'set' object does not support indexing
```

La indexación no funciona para este tipo de dato.

La existencia de un elemento se puede comprobar utilizando la palabra clave 'in'

```
'India' in country
```

```
True
```

El conjunto puede añadir nuevos elementos utilizando la función 'add'.

```
country.add("Canada")
```

Entonces el conjunto se convierte en:

```
{'Canada', 'India', 'UAE', 'USA'}
```

Cuando el conjunto se imprime, los elementos del conjunto no están ordenados.

```
print(country)
```

```
{'India', 'USA', 'Canada', 'UAE'}
```

Los valores duplicados no están permitidos en los conjuntos. Si el elemento "India" se añade por cualquier otra razón, no se reflejará en el conjunto.

```
country.add("India")
print(country)
```

De nuevo añadiendo India al conjunto anterior, el conjunto será:

```
{'India', 'USA', 'Canada', 'UAE'}
```

Longitud del conjunto calculada por el len (),

```
print(len(country))
```

```
4
```

Los elementos del conjunto pueden ser de cualquier tipo de datos

```
country = {'India',20,True,'UAE',30,False}
```

Una tupla y una lista se pueden convertir en un conjunto llamando al constructor set () y el parámetro será el nombre de la lista o de la tupla:

```
list = ['India',20,True,'UAE',30,False]
new_set = set(list)
print(new_set, type(new_set))
```

```
{False, True, 'UAE', 'India', 20, 30} <class 'set'>
```

4. DICCIONARIOS

Los diccionarios son otro tipo de estructura de datos. Cada elemento del diccionario tiene 2 parámetros: clave y valor.

Así que cada elemento del diccionario es un par clave-valor separado por dos puntos:

clave:valor

A cada valor le corresponde una clave.

Sintaxis:

Nombre_diccionario : { clave1: valor1;

clave2:valor2;

…… }

Cada par clave-valor está separado por un punto y coma. Los elementos están encerrados en llaves.

Ejemplo:

```
emails = { "xyz" : "xyz@gmail.com",
           "qas" : "qas@yahoo.com",
           "qwe" : "qwe@gmail.com"}
```

Aquí las claves son xyz, qas, qwe y los valores son xyz@gmail.com, qas@gmail.com, qwe@gamil.com respectivamente.

```
print(emails)
```

```
{'xyz': 'xyz@gmail.com', 'qas': 'qas@yahoo.com', 'qwe': 'qwe@gmail.com'}
```

Se puede acceder a los valores mediante las claves correspondientes

```
emails["xyz"]
```

```
'xyz@gmail.com'
```

El tipo se puede comprobar mediante type ()

```
print(type(emails))
```

```
<class 'dict'>
```

```
print(len(emails))
```

```
3
```

Un elemento existente puede ser modificado especificando la clave. Si el valor qas@gmail.com se quiere modificar en modifyqas@gmail.com, el nuevo valor modificado se puede asignar a la clave correspondiente:

```
emails['qas'] = 'modifyqas@gmail.com'
print(emails)
```

Entonces el diccionario se convierte en,

```
{'qas': 'modifyqas@gmail.com', 'qwe': 'qwe@gmail.com', 'xyz': 'xyz@gmail.com'}
```

Se puede añadir un nuevo elemento al diccionario asignando un nuevo valor a una nueva clave como:

```
emails['abc'] = 'abc@yahoo.com'
print(emails)
```

Entonces el nuevo diccionario se convierte en

```
{'xyz': 'xyz@gmail.com', 'qas': 'qas@yahoo.com', 'qwe': 'qwe@gmail.com', 'abc': 'abc@yahoo.com'}
```

Se puede eliminar un elemento con la palabra clave 'del' especificando únicamente la clave

```
del emails['qas']
print(emails)
```

Entonces el diccionario se convierte en,

```
{'xyz': 'xyz@gmail.com', 'qwe': 'qwe@gmail.com', 'abc': 'abc@yahoo.com'}
```

Los diccionarios no permiten duplicar los valores de los elementos. Si se vuelve a dar un valor a una clave, el valor antiguo se sobrescribe con el nuevo.

```
emails = { "xyz" : "xyz@gmail.com",
           "qas" : "qas@yahoo.com",
           "qwe" : "qwe@gmail.com",
           "xyz" : "xxx@gmail.com"}
```

Aquí se dan dos valores a la misma clave xyz. La segunda vez el valor se sobrescribe por xxx@gmail.com. Así que finalmente se convertirá en el valor de xyz.

```
print(emails)
```

```
{'xyz': 'xxx@gmail.com', 'qas': 'qas@yahoo.com', 'qwe': 'qwe@gmail.com'}
```

En los diccionarios, los pares clave-valor pueden tener cualquier tipo de datos

```
dict_name = { "name" : "xyuz",
              "year": 1876,
              "bool": True,
               1 : ['red','green']}
```

Hay 3 funciones principales que se pueden aplicar al diccionario.

keys () - devuelve una lista de claves de un diccionario

```
print(emails.keys())
```

```
dict_keys(['xyz', 'qas', 'qwe'])
```

values() - devuelve la lista de valores de un diccionario

```
print(emails.values())
```

```
dict_values(['xxx@gmail.com', 'qas@yahoo.com', 'qwe@gmail.com'])
```

items() - devuelve el conjunto de todos los pares clave-valor

```
print(emails.items())
```

```
dict_items([('xyz', 'xxx@gmail.com'), ('qas', 'qas@yahoo.com'), ('qwe', 'qwe@gmail.com')])
```

Si queremos recorrer el diccionario para realizar varias acciones, se puede realizar como:

```
for key,value in emails.items():
  print(key,value)
```

```
xyz xxx@gmail.com
qas qas@yahoo.com
qwe qwe@gmail.com
```

5. ENTRADA/SALIDA DE DATOS

Existen diferentes funciones incorporadas para realizar las operaciones de entrada/salida de datos en python

5.1 Entrada de datos

En python los datos pueden ser introducidos directamente por el operador de asignación

```
[1]  a = 67
     b = 3.224
     c = 3 + 4j
```

U otra opción para introducir los datos se da desde el teclado.

La función *raw_input ()* se utiliza para leer un carácter o cadena y números desde el teclado. Siempre devuelve una cadena incluso si es un número.

Para que sea cogido como un número, debe ser convertido a int usando *int()* o hay una función similar *input()* que puede ser usada para devolver elementos con su tipo de datos apropiado.

Ejemplo:

```
number = raw_input("Enter a number:")
```

Le pedirá al usuario que introduzca un número, si se introduce 2 se muestra

```
Enter a number:2
```

Comprobamos el tipo de variable

```
print(type(number))
```

```
<class 'str'>
```

Como Podemos ver en el ejemplo anterior, el número leído por pantalla es tomado como un string, por lo que debe ser convertido a int para poder operar con él. En lugar de eso input () puede ser usado de la siguiente manera:

```
number = input("Enter a number:")
```

```
Enter a number:2
```

Entonces el tipo de número será int.

5.2 Salida de datos

La función print() puede ser usada para imprimir los datos. Se puede imprimir una cadena o un número de la siguiente manera:

```
print('number')
```

```
number
```

```
print(123)
```

```
123
```

Si un valor de la variable quiere salir con una frase:

```
num = 12
print("The valu of num is:", num)
```

```
The valu of num is: 12
```

La cadena se puede dar entre comillas simples o dobles y el nombre de la variable separado por una coma.

Los 2 parámetros opcionales de print son:

- sep - se utiliza para definir los separadores entre las cadenas separadas por coma

```
print('number',123,'num', sep = "...")
```

```
number...123...num
```

- end - se utiliza cuando la cadena termina con una nueva línea y queremos evitarlo. Normalmente para la función print, si se dan 2 funciones print línea a línea se imprimirá en nueva línea:

```
print("hello")
print("name")
```

```
hello
name
```

Cuando añadimos end a las sentencias print:

```
print("hello",end = " ")
print("name",end = " ")
```

```
hello name
```

Otro método para la salida de los valores de las variables es usar el operador de módulo, %. Hay muchos formatos para diferentes tipos de datos que pueden ser dados con el operador modulo.

Algunas cadenas formateadas son:

- %d - int
- %s - cadena
- %f - float

Las cadenas formateadas se pueden incluir en la función print con el operador de módulo:

print('cadena_formateada' % valor)

```
num = 12
print('number: %d' % num)
```

```
number: 12
```

Del mismo modo, se pueden dar muchos formatos diferentes en una sentencia de impresión.

El orden de la cadena formateada y del valor debe mantenerse adecuadamente.

```
num =12
string = "name"
point = 0.3

print('%d %s %f' % (12,'name',0.3))
```

```
12 name 0.300000
```

Si se necesitan 2 puntos flotantes para un número flotante, entonces se puede formatear como %0.2f.

Del mismo modo, si se necesitan 3 decimales, se establece como %0.3f

```
num = 3.45678
print("%0.3f" % num)
```

```
3.457
```

Los otros tipos de formato son:

`d, i, u`	Decimal integer
`x, X`	Hexadecimal integer
`o`	Octal integer
`f, F`	Floating point
`e, E`	Exponential
`g, G`	Floating point or Exponential
`c`	Single character
`s, r, a`	String
`%`	Single '%' character

6. PUNTOS CLAVE

- En Python tenemos 3 principales controladores de flujo: ***if, for y while***
- Las **listas**, **tuplas**, **diccionarios** y **conjuntos** (**set**) son estructuras que permiten trabajar con colecciones de datos.
- Una **tupla** permite tener agrupados un conjunto inmutable de elementos.
- Los **diccionarios** son objetos que contienen una lista de parejas de elementos.
- Un **conjunto** es una lista de elementos donde ninguno de ellos está repetido.
- Mediante las funciones **print** e **input** podemos mostrar y pedir datos por pantalla.